A study of Convolutions for Efficient Inference of Deep Neural Networks on Embedded Processors

Enrique GALVEZ

Under supervision of:
Alix MUNIER
Adrien CASSAGNE

LIP6, ALSOC Team

February - July 2024

Context and motivations

Convolutionnal Neural Networks (CNNs)

- State-of-the-Art method for most image-based tasks (classification, object-detection...)
- Composed by a succession of layers: Convolution, pooling, activation...
- 3 steps: Design, learning and inference

Context and motivations

Convolutionnal Neural Networks (CNNs)

- State-of-the-Art method for most image-based tasks (classification, object-detection...)
- Composed by a succession of layers: Convolution, pooling, activation...
- 3 steps: Design, learning and inference

Systems-on-Chip (SoCs)

- Good target for CNN inference
- Both latency and energy consumption should be optimized

Context and motivations

Convolutionnal Neural Networks (CNNs)

- State-of-the-Art method for most image-based tasks (classification, object-detection...)
- Composed by a succession of layers: Convolution, pooling, activation...
- 3 steps: Design, learning and inference

Systems-on-Chip (SoCs)

- Good target for CNN inference
- Both latency and energy consumption should be optimized

Contribution

- Study of 4 convolution approaches: `direct`, `im2row`/`im2col`, `winograd` and implicit lowering
- Efficient implementation of these algorithms for CPU inference of usual CNNs
- Performance evaluation with respect to Latency and Energy consumption

Overview

Problem scope

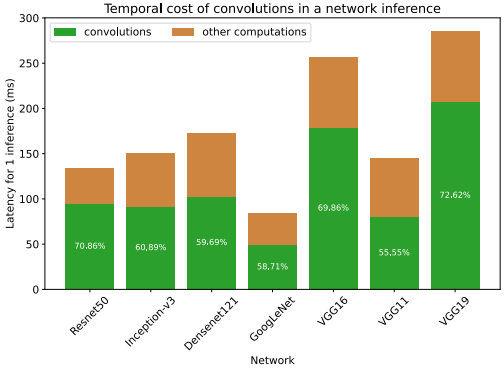


Figure: Proportion of time spent in convolutions for common networks.

Convolution layers:

- Cost almost 70% of the total inference time
- Their implementation is critical for CNN inference

Problem scope

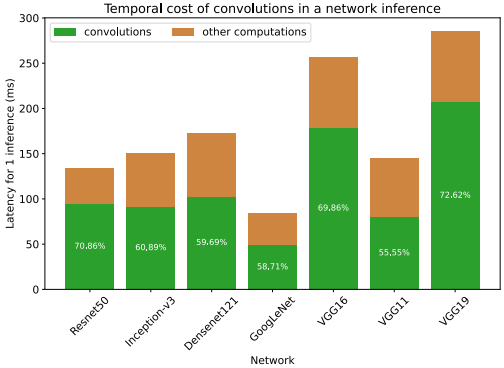


Figure: Proportion of time spent in convolutions for common networks.

Convolution layers:

- Cost almost 70% of the total inference time
- Their implementation is critical for CNN inference

State-of-the-art convolutions:

- Main targets are GPUs/TPUs
- No open-source efficient CPU implementations
- No measures "from the socket" for energy consumption, which is a relevant metric for SoCs

Working with tensors

Tensor: Multidimensional object representing data processed by a CNN

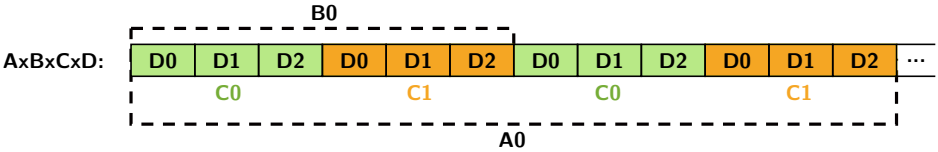


Figure: Tensor with format $A \times B \times C \times D$ in memory.

Working with tensors

Tensor: Multidimensional object representing data processed by a CNN

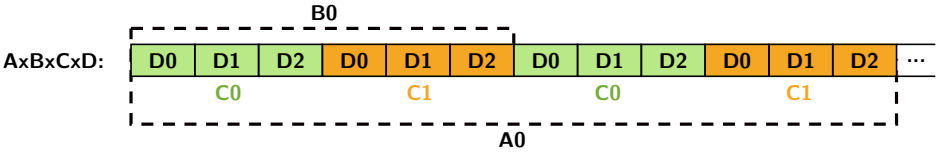


Figure: Tensor with format $A \times B \times C \times D$ in memory.

Image processing tensor: batch $\times$ channels $\times$ image_height $\times$ image_width

Working with tensors

Tensor: Multidimensional object representing data processed by a CNN

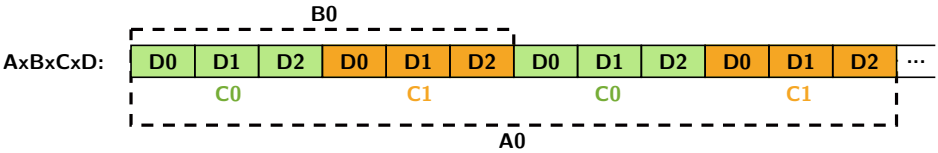


Figure: Tensor with format $A \times B \times C \times D$ in memory.

Image processing tensor: batch $\times$ channels $\times$ image_height $\times$ image_width

MB	batch size	KH, KW	kernel height, width	PH, PW	padding height, width
IC	input channels	OC	output channels	SH, SW	stride height, width
IH, IW	input height, width	OH, OW	output height, width	DH, DW	dilation height, width

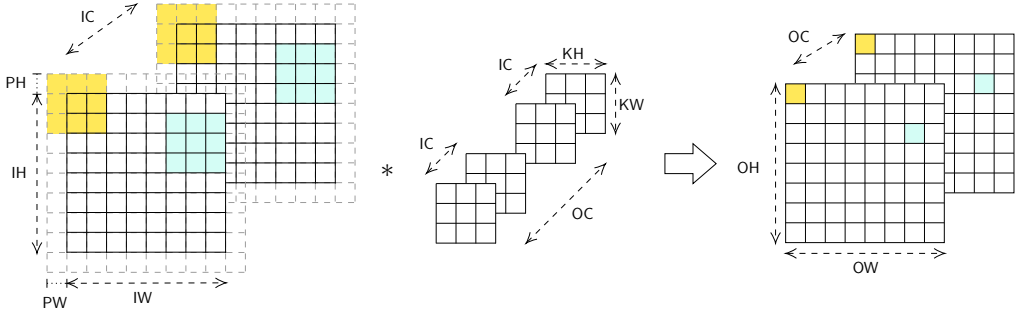
Table: Notations for the main convolution parameters.

Input tensor: $MB \times IC \times IH \times IW$ **Output tensor:** $MB \times OC \times OH \times OW$

The convolution layer

Formula: (Assuming $DH = DW = 0$ and $SH = SW = 1$)

$$dst[mb, oc, oh, ow] = \sum_{ic=0}^{IC-1} \sum_{kh=0}^{KH-1} \sum_{kw=0}^{KW-1} src(mb, ic, \underbrace{oh + kh - PH}_{ih}, \underbrace{ow + kw - PW}_{iw}) \cdot weights[oc, ic, kh, kw].$$



Overview

Direct convolutions

Algorithm 1: Naive direct convolution.

```
1  Input tensor src: (MB × IC × IH × IW)
2  Weights tensor wei: (OC × IC × KH × KW)
3  for mb=0 to MB do
4  for oc=0 to OC do
5  for oh=0 to OH do
6  for ow=0 to OW do
7      d ← 0
8      for ic=0 to IC do
9      for kh=0 to KH do
10     for kw=0 to KW do
11         ih ← oh · SH + kh · (DH + 1) − PH
12         iw ← ow · SW + kw · (DW + 1) − PW
13         d ← d + src[mb, ic, ih, iw] × wei[oc, ic, kh, kw]
14     dst[mb, oc, oh, ow] ← d
15 Return dst: (MB × OC × OH × OW)
```

Naive algorithm:

- Iterates through output tensor
- Sums products of *src* and *wei* elements accross IC, KH, KW

[1] Zhang et al. 2018, "High Performance Zero-Memory Overhead Direct Convolutions"

Direct convolutions

Algorithm 1: Naive direct convolution.

Input tensor src: $(MB \times IC \times IH \times IW)$
Weights tensor wei: $(OC \times IC \times KH \times KW)$
for mb=0 to MB **do**
  for oc=0 to OC **do**
    for oh=0 to OH **do**
      for ow=0 to OW **do**
        $d \leftarrow 0$
        for ic=0 to IC **do**
          for kh=0 to KH **do**
            for kw=0 to KW **do**
              $ih \leftarrow oh \cdot SH + kh \cdot (DH + 1) - PH$
              $iw \leftarrow ow \cdot SW + kw \cdot (DW + 1) - PW$
              $d \leftarrow d + src[mb, ic, ih, iw] \times wei[oc, ic, kh, kw]$
        $dst[mb, oc, oh, ow] \leftarrow d$
Return dst: $(MB \times OC \times OH \times OW)$

Naive algorithm:

- Iterates through output tensor
- Sums products of *src* and *wei* elements accross IC, KH, KW

Optimized version[1]: (See Annex 2)

- Change the order of the loops
- Reuse $src[mb, ih, iw, ic]$ accross OC
- Add cache blocking
- Parallelize accross well-chosen loops
- Use appropriate storage format for *src* and *wei*

[1] Zhang et al. 2018, "High Performance Zero-Memory Overhead Direct Convolutions"

im2row based convolutions

`im2row`[2] algorithm:

- *src* tensor is transformed in a matrix *im_buf*
- The convolution is computed as the matrix multiplication between *M* and *wei*

Main `im2row` benefits:

- GEMM is a very regular operation, allowing hardware and software optimizations
- High performance math libraries provides highly optimized implementations of GEMMs

Algorithm 2: `im2row` convolution.

1 **Input tensor** *src*: $MB \times IH \times IW \times IC$
2 **Weights tensor** *wei*: $(IC \times KH \times KW) \times OC$
3 $im\_buf \leftarrow$ im2row(input) ; // im_buf : $MB \times OH \times OW \times (IC \times KH \times KW)$
4 $dst \leftarrow$ BLAS_GEMM(im_buf, *wei*)
5 **Return** *dst* ; // dst: $MB \times OH \times OW \times OC$

[2] Chellapilla et al. 2006, "High Performance Convolutional Neural Networks for Document Processing"

im2row based convolutions

im2row transformation:

- Data required by each kernel is gathered in a row of the buffer
- Involves data duplication due to overlapping kernels
- Implicit lowering: reduces memory footprint by performing the `im2row` transformation on-the-fly

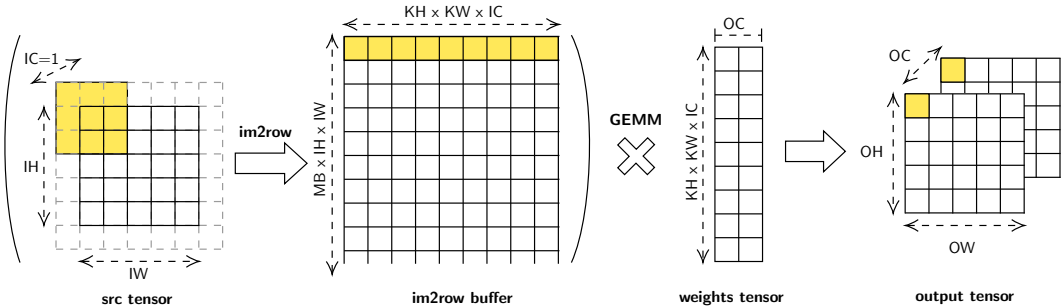


Figure: Computing a convolution using `im2row`.

Winograd's method: example for 1D convolution

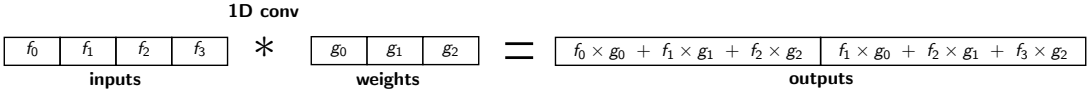


Figure: FIR filter F(2,3) seen as a 1D convolution.

[3] A. Lavin and S. Gray 2015, "Fast Algorithms for Convolutional Neural Networks"

Winograd's method: example for 1D convolution

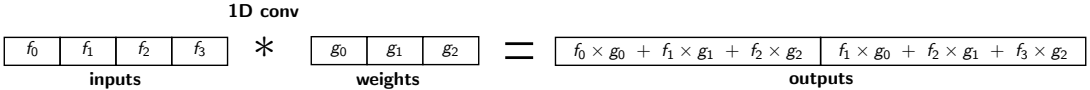


Figure: FIR filter F(2,3) seen as a 1D convolution.

$$Y = \begin{pmatrix} f_0 & f_1 & f_2 \\ f_1 & f_2 & f_3 \end{pmatrix} \begin{pmatrix} g_0 \\ g_1 \\ g_2 \end{pmatrix} = \begin{pmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{pmatrix} \quad \text{with} \quad \begin{array}{ll} m_1 = (f_0 - f_2)g_0, & m_2 = (f_1 + f_2)\dfrac{g_0 + g_1 + g_2}{2}, \\ m_4 = (f_1 - f_3)g_2, & m_3 = (f_2 - f_1)\dfrac{g_0 - g_1 + g_2}{2}. \end{array}$$

Figure: Winograd's algorithm for $F(2,3)$.

[3] A. Lavin and S. Gray 2015, "Fast Algorithms for Convolutional Neural Networks"

Winograd's method: example for 1D convolution

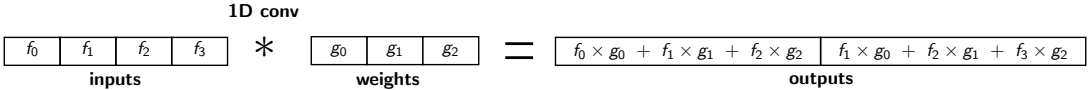


Figure: FIR filter F(2,3) seen as a 1D convolution.

$$Y = \begin{pmatrix} f_0 & f_1 & f_2 \\ f_1 & f_2 & f_3 \end{pmatrix} \begin{pmatrix} g_0 \\ g_1 \\ g_2 \end{pmatrix} = \begin{pmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{pmatrix} \quad \text{with} \quad \begin{aligned} m_1 &= (f_0 - f_2)g_0, \quad m_2 = (f_1 + f_2)\frac{g_0 + g_1 + g_2}{2}, \\ m_4 &= (f_1 - f_3)g_2, \quad m_3 = (f_2 - f_1)\frac{g_0 - g_1 + g_2}{2}. \end{aligned}$$

Figure: Winograd's algorithm for $F(2,3)$.

Winograd's method benefits:

- Default algorithm for $F(m,r)$ requires $m \times r$ multiplications
- Winograd's algorithm requires $m + r - 1$ multiplications for $F(m,r)$
- Number of mutliplications is reduced at the cost of more additions
- Winograd's method can be generalized to 2D convolutions[3] (See Annex 3)

[3] A. Lavin and S. Gray 2015, "Fast Algorithms for Convolutional Neural Networks"

Overview

Measurement platform

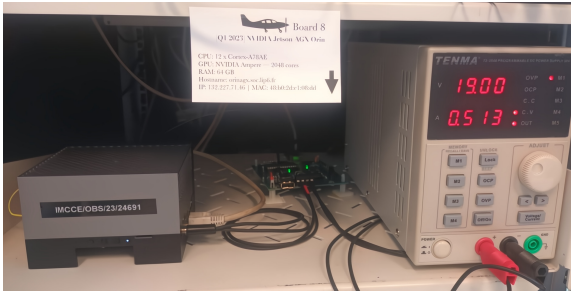


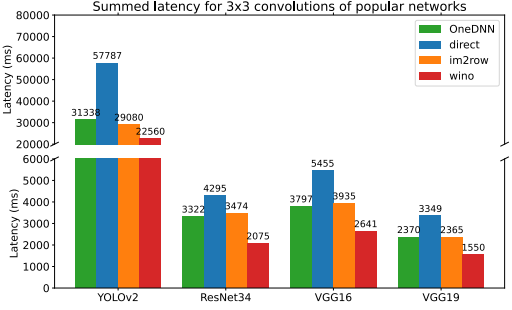
Figure: NVIDIA Jetson AGX Orin.

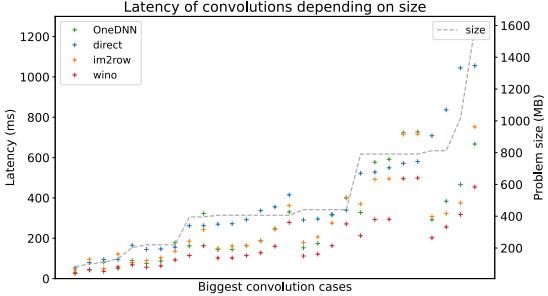
Target SoC: NVIDIA Jetson AGX Orin

- 12× ARM Cortex-A78AE CPU, 64 GB RAM
- Caches: L1 (64kB) and L2 (256kB) in each PU ; L3 (2048kB) shared between 4 PUs

Power and energy consumption: precisely measured from power supply

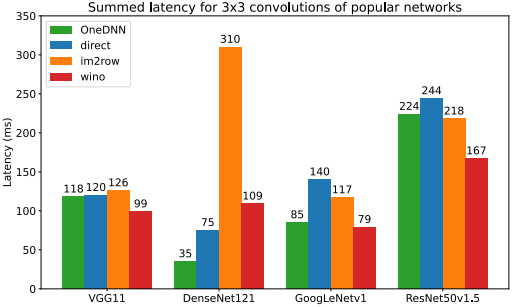
Single thread latency (big layers)

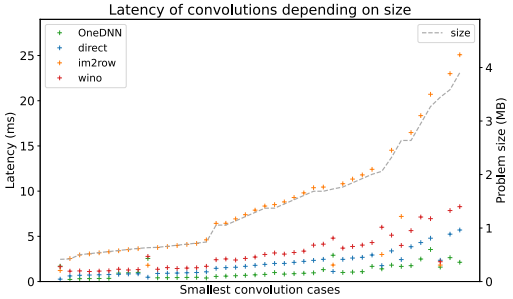




- `wino` is the best implementation on bigger layers
- Lowering-based implementations (`implicit`, `im2row`) are also good
- `direct` suffers from poor performance due to the irregularity of the computations

Single thread latency (small layers)





- `direct` and `implicit` are the best implementation on smaller layers
- `im2row` transform can be very expensive on small convolutions
- `wino` complexity reduction is not enough to compensate transformations cost

Latency compared to State-of-the-Art

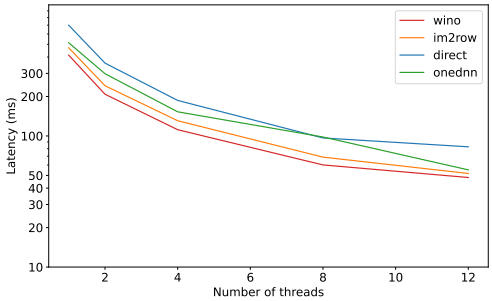


Figure: Latency of ResNet50v1.5 inference depending on parallelism

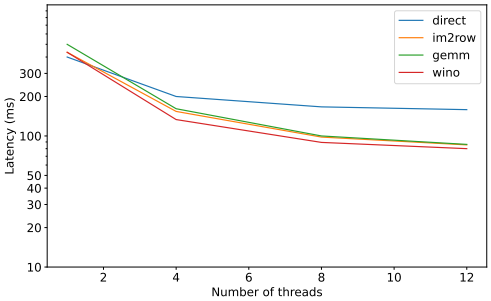


Figure: Latency of ResNet50v1.5 inference depending on parallelism (SotA)

- Similar problem studied in State-of-the-Art on the same target [4]
- We achieve similar results, which validate our methodology

[4] S. Barrachina et al. 2023, "Performance–energy trade-offs of deep learning convolution algorithms on ARM processors"

Energy consumption compared to State-of-the-Art

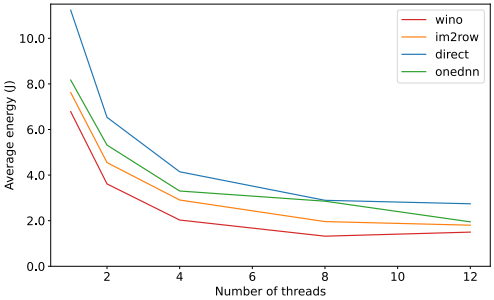


Figure: Energy consumption of ResNet50v1.5 convolutions depending on parallelism

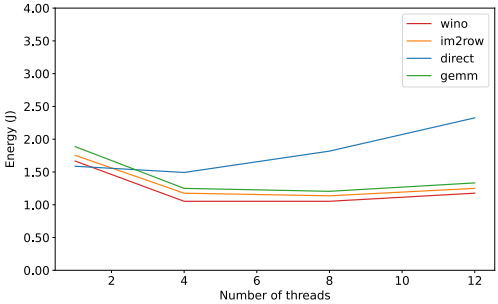


Figure: Energy consumption of ResNet50v1.5 convolutions depending on parallelism (SotA)

- Our measure "from the socket" provide a more relevant information about energy consumption
- SotA measure uses hardware counters only considering CPU and RAM consumption
- Our measure with hardware counters gave similar results to SotA (See Annex 4)

Overview

Conclusion

Several methods for forward convolutions

- `direct`: Straight-forward method, good on small layers
- `im2row/im2col`: Uses a GEMM, big memory overhead, good on biggest layers
- `implicit`: Performs "on-the-fly" im2row to reduce memory overhead
- `winograd`: Reduces arithmetic complexity, use GEMMs, good on big layers

Conclusion

Several methods for forward convolutions

- `direct`: Straight-forward method, good on small layers
- `im2row/im2col`: Uses a GEMM, big memory overhead, good on biggest layers
- `implicit`: Performs "on-the-fly" im2row to reduce memory overhead
- `winograd`: Reduces arithmetic complexity, use GEMMs, good on big layers

Implementing CNN inference on SoCs

- On Jetson AGX Orin, latency and energy consumption are closely related
- Optimal latency is obtained by a compromise between `implicit` and `winograd`
- Measures "form the socket" lead to different conclusions than hardware counters

Conclusion

Several methods for forward convolutions

- `direct`: Straight-forward method, good on small layers
- `im2row/im2col`: Uses a GEMM, big memory overhead, good on biggest layers
- `implicit`: Performs "on-the-fly" im2row to reduce memory overhead
- `winograd`: Reduces arithmetic complexity, use GEMMs, good on big layers

Implementing CNN inference on SoCs

- On Jetson AGX Orin, latency and energy consumption are closely related
- Optimal latency is obtained by a compromise between `implicit` and `winograd`
- Measures "form the socket" lead to different conclusions than hardware counters

Future work

- Continue the study on other architectures
- Explore cross-layer optimizations

Appendix

Thank you for listening !

5. Appendix
   - Annex 1: Convolutions with padding, stride or dilation
   - Annex 2: Optimized direct convolutions
   - Annex 3: Winograd's method generalized to 2D-convolution
   - Annex 4: More performance results

Overview

5. Appendix
   - Annex 1: Convolutions with padding, stride or dilation
   - Annex 2: Optimized direct convolutions
   - Annex 3: Winograd's method generalized to 2D-convolution
   - Annex 4: More performance results

Annex 1: Convolutions with padding, stride or dilation

General formula:

$$dst[mb, oc, oh, ow] = bias[oc] + \sum_{ic=0}^{IC-1} \sum_{kh=0}^{KH-1} \sum_{kw=0}^{KW-1} src(mb, ic, ih, iw) \cdot weights[oc, ic, kh, kw], \quad (1)$$

with:

$$\begin{cases} ih := oh \cdot SH + kh \cdot (DH + 1) - PH, \\ iw := ow \cdot SW + kw \cdot (DW + 1) - PW. \end{cases} \quad (2)$$

Parameters considered: (for simplicity)

- No bias
- Stride and dilation: $SH = SW = 1$ and $DH = DW = 0$
- Kernel size: $KH = KW = 3$

Annex 1: Convolutions with padding, stride or dilation

Formula: (*ih* and *iw* are locally defined)

$$dst[mb, oc, oh, ow] = bias[oc] + \sum_{ic=0}^{IC-1} \sum_{kh=0}^{KH-1} \sum_{kw=0}^{KW-1} src(mb, ic, ih, iw) \cdot weights[oc, ic, kh, kw],$$

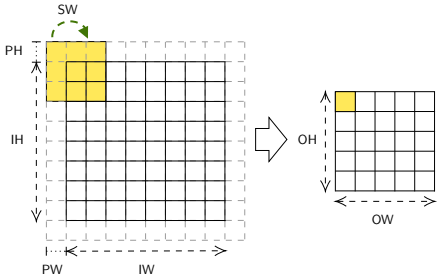


Figure: 2-strided 3×3 convolution.

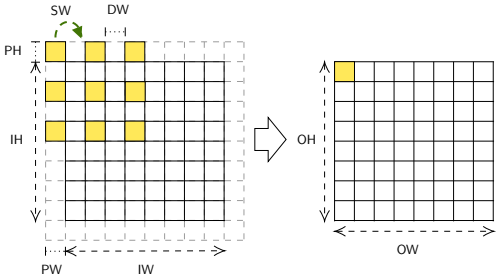


Figure: Dilated 3×3 convolution: $DW = DH = 1$.

Annex 2: Optimized direct convolutions

Algorithm 3: Optimized direct convolution.

1 **Input tensor** *src*: $MB \times IH \times IW \times IC$
2 **Weights tensor** *wei*: $\lceil OC/OC_b \rceil \times \lceil IC/IC_b \rceil \times KH \times KW \times IC_b \times OC_b$
3 **for** mb=*0* **to** MB **do**
4 **for** $oc_b = 0$ **to** $\lceil OC/OC_b \rceil$ **do**
5 **for** ow_b=*0* **to** $\lceil OW/OW_b \rceil$ **do**
6 **for** oh=*0* **to** OH **do**
7 **for** ic_b=*0* **to** $\lceil IC/IC_b \rceil$ **do**
8 **for** kh=*0* **to** KH **do**
9 **for** kw=*0* **to** KW **do**
10 $ih \leftarrow oh \cdot SH + kh \cdot (DH + 1) - PH$
11 **for** ic=$ic_b \times IC_b$ **to** $(ic_b + 1) \times IC_b$ **do**
12 **for** ow=$ow_b \times OW_b$ **to** $(ow_b + 1) \times OW_b$ **do**
13 $iw \leftarrow ow \cdot SW + kw \cdot (DW + 1) - PW$
14 $s \leftarrow src[mb, ih, iw, ic]$
15 **for** oc=$oc_b \times OC_b$ **to** $(oc_b + 1) \times OC_b$ **do**
16 $w \leftarrow wei[oc, ic, kh, kw]$
17 $d \leftarrow s \times w$
18 $dst[mb, oc, oh, ow] \leftarrow dst[mb, oc, oh, ow] + d$
19 **Return** *dst*: $MB \times OH \times OW \times OC$

Loop ordering:

- $MB, OH, KH, KW, IC, OW, OC$
- Reuses $src[mb, ih, iw, ic]$ accross OC

Cache blocking:

- OC, IC and OW are blocked
- *wei* tensor blocked, not *src*
- Allow better reuse of cached data

Parallelism:

- Loops from line 3 to 6 collapsed and parallelized with OpenMp
- IC, KH, KW loops should be executed with appropriate order

Annex 3: Winograd's method generalized to 2D-convolution

Matricial expression:

$$B^T = \begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix}, \qquad G = \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{pmatrix}, \qquad A^T = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{pmatrix},$$

$$g = \begin{pmatrix} g_0 & g_1 & g_2 \end{pmatrix}^T, \qquad d = \begin{pmatrix} f_0 & f_1 & f_2 & f_3 \end{pmatrix}^T, \qquad Y = \text{result tensor}.$$

1D Winograd's method:

- Formula: $Y = A^T\left[(Gg) \odot (B^T d)\right]$
- Number of multiplications for F(m, r): $m + r - 1$

2D Winograd's method:

- Formula: $Y = A^T\left[(GgG^T) \odot (B^T dB)\right]A$
- Number of multiplications for F($m \times m$, $r \times r$): $(m + r - 1)^2$

Annex 3: Winograd's method generalized to 2D-convolution

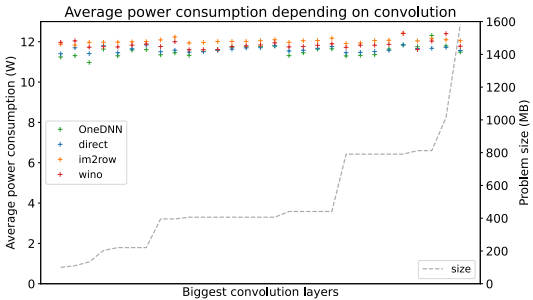
Idea of the algorithm:

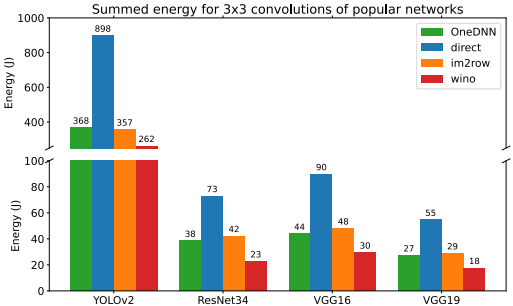
- Input IH and IW dimensions are divided in tiles of size $m + r - 1$
- Corresponding output tiles are computed using 2D Winograd's method on $\mathrm{F}(m \times m,\ r \times r)$
- Use 2D Winograd's formula on each tile: $Y = A^T\left[(GgG^T) \odot (B^T dB)\right]A$

Algorithm 4: Winograd's convolution using $F(m \times m, r \times r)$.

1 d: image tiles, g: weights of kernels, Y: output tiles
2 G, B^T, A^T: static transformation matrices
3 $\alpha^2 = (m + r - 1)^2$: size of an input tile, P: number of tiles
4 `// transform weights with additions`
5 $U[:, :, oc, ic] \leftarrow G \cdot g[oc, ic, :, :] \cdot G^T$; // $U : \alpha \times \alpha \times OC \times IC$
6 `// transform input tensor with additions`
7 $V[:, :, ic, b] \leftarrow B^T \cdot d[ic, b, :, :] \cdot B$; // $V : \alpha \times \alpha \times IC \times P$
8 `// compute multiplications using a GEMM`
9 $M[\xi, \nu, :, :] \leftarrow BLAS\_GEMM(U[\xi, \nu, :, :], V[\xi, \nu, :, :])$; // $M : \alpha \times \alpha \times OC \times P$
10 `// transform output with additions`
11 $Y[oc, b, :, :] \leftarrow A^T \cdot y[:, :, oc, b] \cdot A$; // $Y : OC \times P \times m \times m$
12 **Return** Y

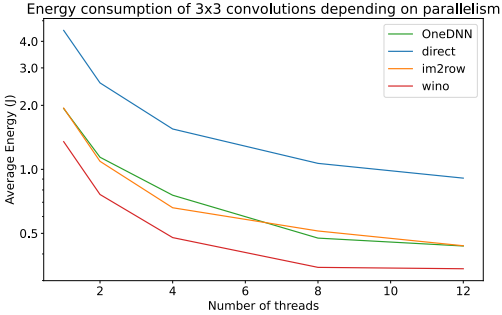
Annex 4: More performance results

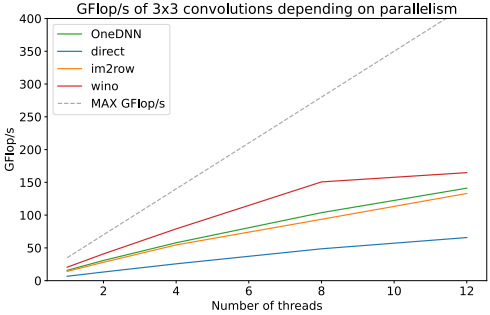




- Energy consumption is computed as: $Energy = Power \times Latency$
- Power consumption is relatively stable depending on the implementation
- Energy consumption is thus determined by latency

Annex 4: More performance results





- With increasing number of threads, latency decreases faster than instantaneous power increases
- As a result, energy consumption decreases when the number of threads increases
- Because `wino` tiles have size 4×4, it has poor scalability for 12 threads

Annex 4: More performance results

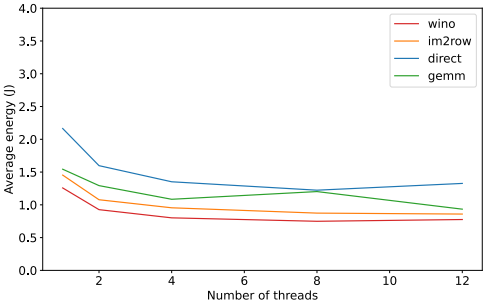


Figure: Energy consumption of ResNet50v1.5 convolutions depending on parallelism

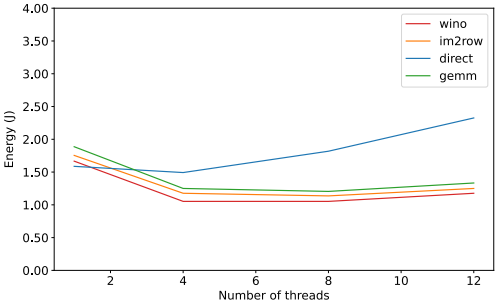


Figure: Energy consumption of ResNet50v1.5 convolutions depending on parallelism (SotA)

- SotA measure uses hardware counters only considering CPU and RAM consumption
- Our measures with hardware counters gave similar results to SotA